DOI:10.1145/2330667.2330691

CryptDB: Processing Queries on an Encrypted Database

By Raluca Ada Popa, Catherine M.S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan

1. INTRODUCTION

Theft of private information is a significant problem for online applications. For example, a recent investigation found that at least eight million people's medical records were stolen as a result of data breaches between 2009 and 2011,[13] and in a recent attack on the Sony Playstation Network, attackers apparently gained access to about 77 million personal user profiles, some of which included credit card information.[20] Such large-scale data thefts make the popular press, but smaller-scale compromises occur on a nearly daily basis, according to organizations devoted to studying consumer and data privacy (e.g., the Privacy Rights Clearinghouse).

Sensitive data can leak from online data repositories for a variety of reasons: an adversary can exploit software vulnerabilities to gain unauthorized access to servers,[15] curious or malicious administrators at a hosting provider can snoop on private data,[3] and attackers with physical access to servers can steal data from disk and memory.[11]

One approach to reduce the damage caused by server compromises is to encrypt all sensitive data stored on the servers. However, many important applications, including database-backed Web services that process SQL queries, as well as analytic applications that compute results over large quantities of data, require servers to not just store data, but also perform computations on the data. One solution could be to store the data encrypted at the server, but to perform all computation at a trusted client on plaintext by downloading and decrypting all needed data for every computation; this approach, however, is usually untenable because there might be too much data to move around, or because clients may have significantly less computation or storage resources than the server.

An ideal solution to satisfying the dual goals of protecting data confidentiality and running computations is to enable a server to *compute over encrypted data*, without the server ever decrypting the data to plaintext. The server would produce results in an encrypted form, decryptable only by a trusted client. This approach would preserve the architecture of running much of the application's computation at the server.

Theoretical approaches such as fully homomorphic encryption[7] enable the server to compute arbitrary functions over encrypted data, while providing excellent confidentiality guarantees. But despite good progress in recent years, these schemes remain many orders of magnitude slower than equivalent plaintext computations (e.g., computing the decryption circuit for AES—the Advanced Encryption Standard[6] is at least 10^9 times slower[8]).

We introduce CryptDB, a practical system that explores an intermediate design point to provide confidentiality for applications that use database management systems (DBMSes). CryptDB is the first practical system that can *execute a wide range of SQL queries over encrypted data*. The key insight that makes our approach practical is that most SQL queries use a small set of well-defined operators, each of which we are able to support efficiently over encrypted data.

CryptDB addresses two threats, as illustrated in Figure 1. The first threat is an adversary who gains access to the DBMS server and tries to learn private data (e.g., health records, financial statements, and personal information) by snooping on the server. This threat might arise when an attacker exploits some vulnerability to directly get to the DB server, when the database is outsourced to an external organization (e.g., a public "cloud"), or when the DBMS is administered by a curious system or database administrator (DBA) who might not be trusted. CryptDB aims to prevent the adversary from learning private data in this case. The second threat is an adversary who gains complete control of the application and the DBMS servers. In this case, CryptDB protects the confidentiality of the data belonging only to users logged-out of the application during an attack, but cannot provide any guarantees for logged-in users. This paper focuses primarily on the solution to the first threat; our SOSP paper[18] details the additional mechanisms that address the second threat.

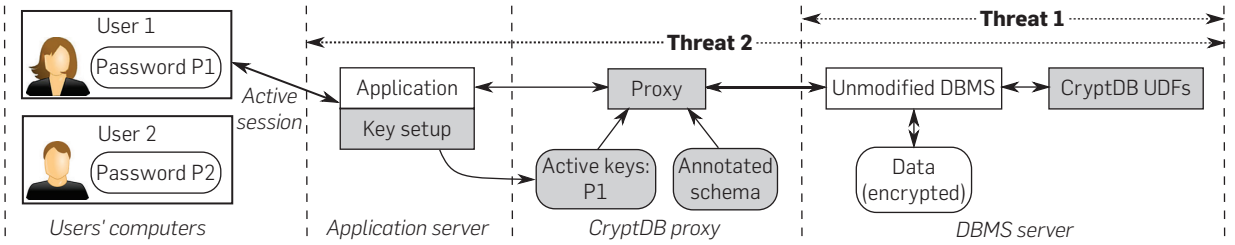
CryptDB requires no changes to the internals of the DBMS server, and should work with most standard SQL DBMSes. Our implementation uses a MySQL back-end. Our experiments show that the overhead of CryptDB is modest: throughput reduces by only 26% for queries from the standard TPC-C benchmark, and by only 14.5% for a multiuser bulletin board application (phpBB),[18] compared to running them over MySQL without encryption. We find that CryptDB supports most queries observed in practice: an analysis of 126 million SQL queries from an MIT MySQL service showed that CryptDB supports operations over encrypted data for 99.5% of the 128,840 columns seen in the query trace.

2. THREAT MODEL AND OVERVIEW

In this section, we discuss CryptDB's threat model and provide an overview of our approach.

A previous version of this paper was published in the *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, October 2011.

Figure 1. CryptDB's architecture consisting of two parts: a *proxy* and an unmodified *DBMS*. CryptDB uses user-defined functions (UDFs) to perform cryptographic operations in the DBMS. Rectangular and rounded boxes represent processes and data, respectively. Shading indicates components added by CryptDB. Dashed lines indicate separation between users' computers, the application server, a server running CryptDB's proxy (which is usually the same as the application server), and the DBMS server. The scope of the two threats CryptDB addresses is shown as dotted lines.



2.1. Threat 1: DBMS server compromise

CryptDB provides confidentiality (data secrecy) in the face of an attacker with full read access to the data stored in the DBMS server. The attacker is assumed to be *passive*: she wants to learn confidential data, but does not change queries issued by the application, query results, or the data in the DBMS. This threat includes DBMS software compromises, root access to DBMS machines, and even access to the RAM of physical machines. With the rise in database consolidation inside enterprise data centers, outsourcing of databases to public cloud computing infrastructures, and the use of third-party DBAs, this threat is increasingly important. We focus on confidentiality, not data integrity or availability.

CryptDB addresses this threat by executing SQL queries over encrypted data on the DBMS server. As shown in Figure 1, CryptDB works by intercepting all SQL queries in a trusted *proxy;* existing applications do not need to be modified to use CryptDB, but all queries must go through the proxy. The proxy stores a master secret key, which it uses to rewrite queries to execute on encrypted data. The proxy encrypts and decrypts all data, and changes some query operators, while preserving the semantics of the query. Because the DBMS server never receives decryption keys to the plaintext, it never sees sensitive data, ensuring that our passive adversary cannot gain access to private information.

The main challenge when executing queries on encrypted data lies in the tension between minimizing the amount of confidential information revealed to the DBMS server and the ability to efficiently execute a variety of queries. Our strategy is to allow the DBMS server to perform query processing on encrypted data *mostly as it would on an unencrypted database* (important for practicality), while restricting the server to computing *only the functions required to process authorized queries* (important for confidentiality). For example, if the DBMS needs to perform a `GROUP BY` on column *c*, the DBMS server should be able to determine which items in that column are equal to each other, but not the actual content of each item. Therefore, the proxy needs to enable the DBMS server to determine relationships among data items necessary to process a query.

CryptDB is careful about what relations between tuples it reveals to the DBMS server. To execute a `GROUP BY` on column *c*, for instance, the server need not know the order of the items in column *c*, nor any information about other columns. To execute an `ORDER BY`, or to find the `MAX` or `MIN`, CryptDB reveals the order of items in that column, but not otherwise.

CryptDB incorporates two techniques: *SQL-aware encryption* and *adjustable query-based encryption*. SQL-aware encryption uses the observation that most SQL queries are made up of a well-defined set of basic operators, such as equality checks, order comparisons, aggregates (sums), and joins. CryptDB supports these operators over encrypted data. By adapting known encryption schemes (for equality, additions, and order checks), and using a new privacy-preserving cryptographic scheme for joins, CryptDB encrypts each data item in a way that allows the DBMS to execute on the transformed data.

The second technique is *adjustable query-based encryption:* CryptDB carefully *adjusts* the SQL-aware encryption scheme for any given data item to support different operations on this data. To implement these adjustments efficiently, CryptDB uses *onions of encryption*. Onions are a novel way to compactly store multiple ciphertexts within each other in the database and avoid revealing weaker encryption schemes when they are not needed.

CryptDB provides *confidentiality for the content of the data* and for names of columns and tables, but does not hide the overall table structure, the number of rows, the types of columns, or the approximate size of data in bytes. The only information that CryptDB reveals to the DBMS server is relationships among data items corresponding to *classes of computation* that queries perform on the database, such as comparing items for equality, sorting, or performing word search. The granularity at which CryptDB allows the DBMS to perform a class of computations is an entire column (or a group of joined columns, for joins), which means that even if a query requires equality checks for a few rows, executing that query on the server would require revealing that class of computation for an entire column. Section 3.1 describes how these classes of computation map to CryptDB's encryption schemes, and the information they reveal.

CryptDB provides the following properties:

- Sensitive data is never available in plaintext at the DBMS server.
- If the application requests no relational predicate filtering on a column, nothing about the data content leaks (other than its size in bytes). In Section 5, we show that all or almost all of the most sensitive fields in the tested applications are in this category.
- If the application requests equality checks on a column, CryptDB's proxy reveals which items repeat in that column, but not the actual values.
- If the application requests order checks on a column, the proxy reveals the order of the elements in the column.

2.2. Threat 2: arbitrary threats

The approach in threat 1 is insufficient when the application server, the proxy, and the DBMS server infrastructure may all be arbitrarily compromised. The reason is that an adversary corrupting the proxy can now get access to the master key used to encrypt the entire database.

The solution is to use the SQL-aware and adjustable encryption techniques, but not with a single master key. Instead, we use per-user keys, derived from the user's password, each having access to only a subset of the data.

A challenge is that simply encrypting each user's data with that user's password does not work because users may share data. To permit data sharing, we encrypt each data item with a new key, and *chain these new keys to user passwords*, so that each data item can be decrypted only through a chain of keys rooted at the password of a user with legitimate access to that data. To construct a chain of keys that captures the application's data privacy and sharing policy, CryptDB requires the developer to provide policy annotations over the application's SQL schema.

Because queries still execute over encrypted data, the passive adversary of threat 1 remains at bay. In addition, even in the face of *arbitrary* server-side compromises, CryptDB protects the data of users logged out for the duration of an attack, since none of the components compromised by this attack have access to the keys of those users. However, an adversary that compromises the application server or proxy can gain access to data of users logged in during the attack, by obtaining their keys. By "duration of an attack," we mean the interval from the start of a compromise until any trace of the compromise has been erased from the system.

3. QUERIES OVER ENCRYPTED DATA

This section describes how CryptDB executes SQL queries over encrypted data in the face of the threat described in Section 2.1.

The CryptDB proxy stores a secret master key *MK*, the database schema, and the current encryption layer of each column. The DBMS server sees an anonymized schema (in which table and column names are replaced by opaque identifiers), encrypted user data, and some auxiliary tables used by CryptDB. CryptDB also equips the server with certain user-defined functions (UDFs) that enable the server to compute on ciphertexts for certain operations.

Processing a query in CryptDB involves four steps:

1. The application issues a query, which the proxy intercepts and rewrites: it anonymizes each table and column name, and, using the master key *MK*, encrypts each constant in the query with an encryption scheme best suited for the desired operation (Section 3.1). The proxy also replaces certain operations with UDFs.
2. The proxy checks if the DBMS server should be given keys to adjust encryption layers before executing the query, and if so, issues an `UPDATE` query at the DBMS server, which invokes a UDF to adjust the encryption layer of the appropriate columns (Section 3.2).
3. The proxy sends the encrypted query to the server, which executes it.
4. The server returns the encrypted query result, which the proxy decrypts and returns to the application.

3.1. SQL-aware encryption

We now describe the encryption methods used in CryptDB, including a number of existing cryptosystems and a new cryptographic primitive for joins. For each encryption method, we explain the security property that CryptDB requires from it, its functionality, and how it is implemented.

Random (RND). RND provides the maximum security in CryptDB: indistinguishability under an adaptive chosen-plaintext attack (IND-CPA). This scheme is probabilistic, meaning that two equal values are mapped to different ciphertexts with high probability. On the other hand, RND does not allow any computation to be performed efficiently on the ciphertext. An efficient construction of RND is to use a block cipher like advanced encryption standard (AES)[6] or Blowfish in CBC mode together with a random initialization vector (IV). (We mostly use AES, except for integer values, where we use Blowfish for its 64-bit block size because the 128-bit block size of AES would cause the ciphertext to be significantly longer.)

Deterministic (DET). DET enables the server to learn which encrypted values correspond to the same data value, by deterministically generating the same ciphertext for the same plaintext. Therefore, this encryption layer allows the server to perform equality checks, which means it can perform selects with equality predicates, equality joins, `GROUPBY`, `COUNT`, `DISTINCT`, etc.

In cryptographic terms, DET should be a pseudo-random permutation (PRP).[9] We use Blowfish or AES in CMC mode[10] to implement DET.

Order-preserving encryption (OPE). OPE allows the server to determine order relations between data items based on their encrypted values, without revealing the data itself. If $x < y$, then $\mathrm{OPE}_K(x) < \mathrm{OPE}_K(y)$, for any secret key K. Therefore, if a column is encrypted with OPE, the server can perform range queries when given encrypted constants $\mathrm{OPE}_K(c_1)$ and $\mathrm{OPE}_K(c_2)$ corresponding to the range $[c_1, c_2]$. The server can also perform `ORDER BY`, `MIN`, `MAX`, `SORT`, etc.

OPE is a weaker encryption scheme than DET because it reveals order. Thus, the CryptDB proxy will only reveal

OPE-encrypted columns to the server if users request order queries on those columns. OPE is proven to be equivalent to a random mapping that preserves order.[1] However, such a mapping leaks half of the data bits in the worst case.[2] We are currently working on a new scheme that provably reveals only order and leaks no bits in addition.

Homomorphic encryption (HOM). HOM is as secure a probabilistic encryption scheme as RND, but allows the server to perform computations on encrypted data with the final result decrypted at the proxy. Although fully homomorphic encryption is prohibitively slow, homomorphic encryption for specific operations is efficient. To support additions, we implemented the Paillier cryptosystem.[17] With Paillier, multiplying the encryptions of two values results in an encryption of the sum of the values, that is, $\mathrm{HOM}_K(x) \cdot \mathrm{HOM}_K(y) = \mathrm{HOM}_K(x+y)$, where the multiplication is performed modulo some public-key value. To compute `SUM` aggregates, the proxy replaces `SUM` with calls to a UDF that performs Paillier multiplication on a column encrypted with HOM. HOM can also be used to compute averages by having the DBMS server return the sum and the count separately, and to increment values (e.g., `SET id = id + 1`). HOM ciphertexts are 2048 bits long.

Join (JOIN and OPE-JOIN). A separate encryption scheme is needed to allow equality join between two columns, because we use different column-specific keys for DET to prevent correlations between columns. JOIN not only supports all the operations allowed by DET, but also enables the server to determine repeating values between two different columns. OPE-JOIN enables joins by order relations. We provide a new cryptographic scheme for JOIN (Section 3.4).

Word search (SEARCH). SEARCH is used to perform searches on encrypted text to support operations such as MySQL's `LIKE` operator. SEARCH is nearly as secure as RND. We implemented the method of Song et al.[22] SEARCH currently supports only full word searches.

When the user performs a query such as `SELECT * FROM messages WHERE msg LIKE "% alice %"`, the proxy gives the DBMS server a token, which is an encryption of `alice`. The server cannot decrypt the token to figure out the underlying word. Using a user-defined function, the DBMS server checks if any of the word encryptions in any message match the token. All that the server learns from a SEARCH query is whether the token matched a message or not, and only for the tokens requested by the user. The server would learn the same information when returning the result set to the users, so the scheme reveals the minimal amount of additional information needed to return the result.

3.2. Adjustable query-based encryption

Our goal is to use the most secure encryption schemes that enable running the requested queries. For example, if the application issues no queries that compare data items in a column, or that sort a column, the column should be encrypted with RND. For columns that require equality checks but not order checks, DET suffices. The problem is that the query set is not always known in advance. Thus, we need an adaptive scheme that dynamically adjusts encryption strategies.

CryptDB's *adjustable query-based encryption* technique solves this problem by dynamically adjusting the layer of encryption on the DBMS server. The idea is to encrypt each data item in one or more *onions:* that is, each value is dressed in layers of increasingly stronger encryption, as shown in Figures 2 and 3. Each layer of each onion enables a certain class of computation, as explained earlier.

Multiple onions are required because the computations supported by different encryption schemes are not always strictly ordered. Depending on the type of the data, CryptDB may not maintain all onions for each column. For instance, the *Search* onion does not make sense for integers, and the *Add* onion does not make sense for strings.

For each layer of each onion, the proxy uses the same key for encrypting values in the same column, and different keys across tables, columns, onions, and onion layers. Using the same key for all values in a column allows the proxy to perform operations on a column without having to compute separate keys for each row that will be manipulated. Using different keys across columns prevents the server from learning any additional relations. All of these keys are derived from the master key *MK*. For example, for table *t*, column *c*, onion *o*, and encryption layer *l*, the proxy uses the key

$$K_{t,c,o,l} = \mathrm{PRP}_{MK}(\text{table } t, \text{column } c, \text{onion } o, \text{layer } l), \qquad (1)$$

where PRP is a pseudorandom permutation (e.g., AES).

Each onion starts out with the most secure encryption scheme as the top level (RND for onions *Eq* and *Ord*, HOM for onion *Add*, and SEARCH for onion *Search*). As the proxy receives SQL queries from the application, it determines whether layers of encryption need to be removed. If a query requires predicate *P* on column *c*, the proxy first establishes what onion layers are needed to compute *P* on *c*. If the encryption of *c* is not already at an onion layer that allows *P*, the proxy strips off the onion layers to allow *P* on *c*, by sending the corresponding onion key to the server. The proxy never decrypts the data past the least-secure non-plaintext encryption onion layer, which may be overridden by the schema developer to be a more secure layer (e.g., one may

Figure 2. Onion encryption layers and the classes of computation they allow. Onion names stand for the operations they allow at some of their layers (Equality, Order, Search, and Addition). A random IV for RND (Section 3.1), shared by the RND layers in *Eq* and *Ord*, is also stored for each data item.

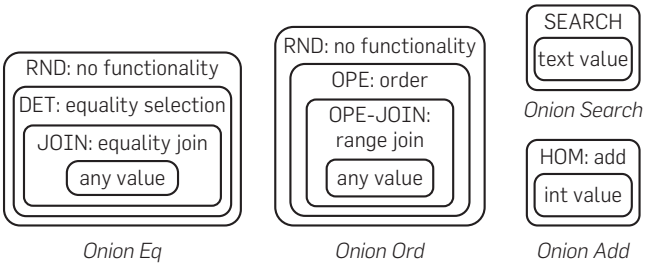
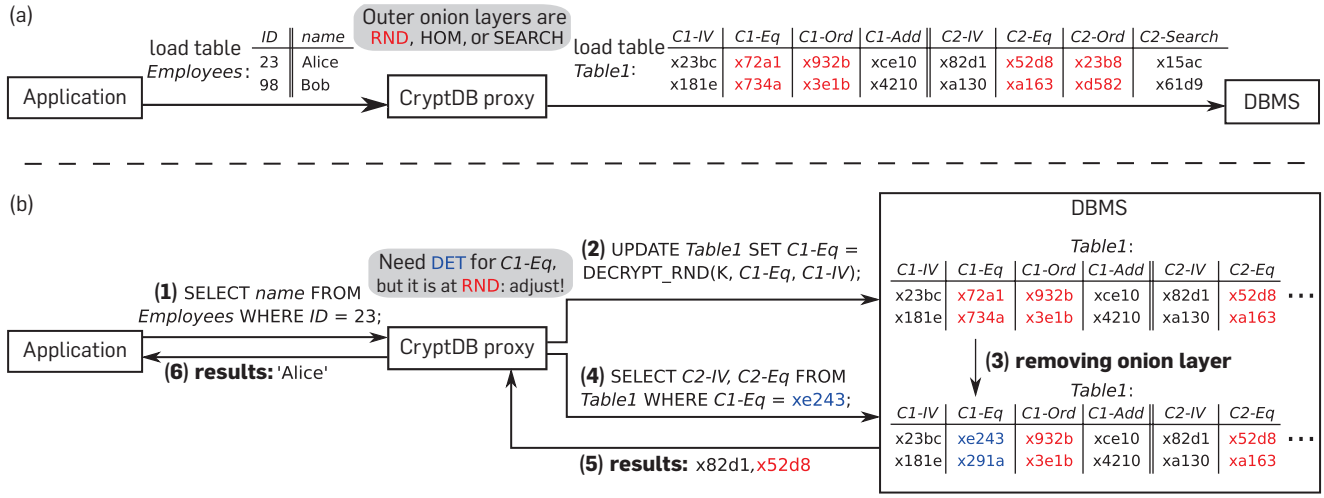


Figure 3. Examples of (a) how CryptDB transforms a table's schema and encrypts a database, and of (b) a query flow showing onion adjustments. Strings of the form "x..." denote ciphertexts (not shown to their full length).



specify that credit card information may at worst be at DET, and never at OPE).

CryptDB implements onion layer decryption using UDFs that run on the DBMS server. For example, in Figure 3, to decrypt onion *Ord* of column 2 in Table 1 to layer OPE, the proxy issues the following query to the server, invoking the `DECRYPT_RND` UDF:

```
UPDATE Table1 SET
C2-Ord = DECRYPT_RND(K, C2-Ord, C2-IV,)
```

where K is the appropriate key computed from Equation (1). At the same time, the proxy updates its own internal state to remember that column *C2-Ord* in *Table1* is now at layer OPE in the DBMS.

Note that onion decryption is performed entirely by the DBMS server. In the steady state, no server-side decryptions are needed, because onion decryption happens only when a new class of computation is requested on a column. For example, after an equality check is requested on a column and the server brings the column to layer DET, the column remains in that state, and future queries with equality checks require no decryption. This property is the main reason why CryptDB's run-time overhead is modest (Section 5).

3.3. Executing over encrypted data

Once the onion layers in the DBMS are at the layer necessary to execute a query, the proxy transforms the query to operate on these onions. In particular, the proxy replaces column names in a query with corresponding onion names, based on the class of computation performed on that column. For example, for the schema shown in Figure 3, a reference to the *Name* column for an equality comparison will be replaced with a reference to the *C2-Eq* column.

The proxy also replaces each constant in the query with a corresponding onion encryption of that constant, based on the computation in which it is used. For instance, if a query contains `WHERE Name = "Alice"`, the proxy encrypts "Alice" by successively applying all encryption layers corresponding to onion *Eq* that have not yet been removed from *C2-Eq*.

Finally, the proxy replaces certain operators with UDF-based counterparts. For instance, the `SUM` aggregate operator and the + column-addition operator must be replaced with an invocation of a UDF that performs HOM addition of ciphertexts. Equality and order operators (such as = and <) do not need such replacement and can be applied directly to the DET and OPE ciphertexts.

Read query execution. To understand query execution over ciphertexts, consider the example schema shown in Figure 3(a). Initially, each column in the table is dressed in all onions of encryption, with RND, HOM, and SEARCH as outermost layers, as shown in Figure 2. At this point, the fields are protected with strong encryption schemes. Figure 3(b) then shows an example of processing an equality predicate on the encrypted data. This query (step 1) requires a lower onion layer for execution than the one present in the DBMS, so the proxy removes this layer at the server using the UPDATE query in (2) by invoking the decryption UDF. Column *C1* corresponds to *ID*, and `xe243` is the *Eq* onion encryption of "23" with keys $K_{T1,C1,Eq,\text{JOIN}}$ and $K_{T1,C1,Eq,\text{DET}}$ (see Figure 2). After the DB server processes the adjustment in (3), the proxy issues the transformed select query (4), and receives encrypted results (5). Note that the proxy must request the random IV from column `C2-IV` in order to decrypt the RND ciphertext from `C2-Eq`. Finally, the proxy decrypts the results from the server using keys $K_{T1,C2,Eq,\text{RND}}$, $K_{T1,C2,Eq,\text{DET}}$, and $K_{T1,C2,Eq,\text{JOIN}}$, obtains the result "Alice," and returns it to the application (6).

Write query execution. CryptDB supports `INSERT`, `DELETE`, and `UPDATE` queries in a similar way to `SELECT`. An `UPDATE` of a column value based on an existing column value, such as `salary = salary + 1`, is more involved.[18]

3.4. Computing joins

There are two kinds of joins supported by CryptDB: *equi-joins*, in which the join predicate is based on equality, and *range joins*, which involve order checks. To perform an equi-join of two encrypted columns, the columns should be encrypted with the same key so that the server can see matching values between the two columns. At the same time, to provide better privacy, the DBMS server should not be able to join columns for which the application did not request a join, so columns that are never joined should not be encrypted with the same keys.

If the queries that can be issued, or the pairs of columns that can be joined, are known *a priori*, equi-join is easy to support: CryptDB can use the DET encryption scheme with the same key for each group of columns that are joined together. However, the challenging case is when the proxy does not know the set of columns to be joined *a priori*, and hence does not know which columns should be encrypted with matching keys.

To solve this problem, we introduce a new cryptographic primitive, JOIN-ADJ (*adjustable join*), which allows the DBMS server to adjust the key of each column at runtime. Intuitively, JOIN-ADJ can be thought of as a "keyed random hash" with the additional property that hashes can be adjusted to change their key *without access to the plaintext*. JOIN-ADJ is a deterministic function of its input, which means that if two plaintexts are equal, the corresponding JOIN-ADJ values are also equal. With JOIN-ADJ, initially, each column uses a different key for the JOIN layer, thus preventing any joins between columns. When a query requests a join, the proxy gives the DBMS server an "adjustment" key to adjust the JOIN-ADJ values in one of the two columns (the first column in lexicographic order), so that it matches the JOIN-ADJ key of the other column. After the adjustment, the columns share the same JOIN-ADJ key, allowing the DBMS server to join them for equality (for this or future queries). Our previous publications[18, 19] describe the JOIN-ADJ scheme formally and prove its security guarantees.

For range joins, a similar dynamic readjustment scheme is difficult to construct due to the lack of structure in OPE schemes. Instead, CryptDB requires that pairs of columns that will be involved in such joins be declared by the application ahead of time, so that matching keys are used for layer OPE-JOIN of those columns; otherwise, the same key will be used for all columns at layer OPE-JOIN. Fortunately, range joins are rare; they are not used in any of our example applications, and are used in only 50 out of 128,840 columns in a large SQL query trace we describe in Section 5.

3.5. Other queries and limitations

CryptDB supports most relational queries and aggregates on standard data types, such as integers and text/varchar types. Additional operations can be added to CryptDB by extending its existing onions, or adding new onions for specific data types (e.g., spatial and multidimensional range queries[21]). Alternatively, in some cases, it may be possible to map complex unsupported operations to simpler ones (e.g., extracting the month out of an encrypted date is easier if the date's day, month, and year fields are encrypted separately).

There are certain computations CryptDB cannot support on encrypted data. For example, it does not support order comparison with a summation, such as `WHERE salary > age + 10`. One could support such a query by splitting it into different queries and having the proxy re-encrypt intermediate results.

Most other DBMS mechanisms, such as transactions and indexing, work the same way over encrypted data as they do over plaintext, with no modifications.

4. IMPLEMENTATION

The CryptDB proxy is built on top of `mysql-proxy`, and consists of a C++ library and a Lua module. The C++ library consists of a query parser; a query encryptor/rewriter, which encrypts fields or includes UDFs in the query; and a result decryption module. The query rewriter operates on the abstract syntax tree (AST) of the SQL query. Given an expression, the rewriter produces replacement expressions for the value of the original expression encrypted with different encryption types (e.g., RND, DET, or just "plaintext"). We use NTL and OpenSSL for cryptographic operations. Our prototype consists of ∼18,000 lines of C++ code and ∼150 lines of Lua code, with another ∼10,000 lines of test code. CryptDB is portable; we have implemented versions for both Postgres 9.0[18] and MySQL 5. CryptDB requires only UDF support from the DBMS and does not change the DBMS server software.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate three aspects of CryptDB: what types of queries and applications does CryptDB support, what is the level of security that CryptDB provides, and what is the performance impact of using CryptDB?

We analyze the functionality and security of CryptDB on five applications and one large trace: phpBB (an open-source Web forum application), HotCRP (a conference management system), grad-apply (the MIT EECS graduate admission application), Open-EMR (an electronic medical records application storing patient medical data), TPC-C (an industry-standard database benchmark), and a trace of SQL queries from a popular MySQL server at MIT, sql.mit.edu. This server is used primarily by Web applications running on scripts.mit.edu, a shared Web application hosting service operated by MIT's Student Information Processing Board (SIPB). In addition, this SQL server is used by a number of applications that run on other machines and use sql.mit.edu only to store their data. Our query trace spans about ten days, and includes approximately 126 million queries over 1193 databases and 18,162 queries. Each database is likely to be a separate instance of an application. All these applications and the large SQL trace contain sensitive information that should be protected (e.g., medical records, student grades, and private messages).

In the first four applications (not counting TPC-C and the large trace), we manually identify which columns are likely to be sensitive and encrypt only those. Some fields were clearly sensitive (e.g., grades, private messages, and medical information), but others were only marginally so (e.g., the time at which a message was posted). There was no clear threshold between sensitive or not, but it was clear

to us which fields were definitely sensitive. In the case of TPC-C and the large query trace, we encrypt all the columns in the database to study the performance of a fully encrypted DBMS or understand which queries or columns are not supported.

We also evaluate the overall performance of CryptDB on the phpBB application and present a detailed analysis through microbenchmarks on a query mix from TPC-C.

5.1. Functional evaluation

To evaluate what columns, operations, and queries CryptDB can support, we analyzed the queries issued by the applications described above. The results are shown in the left half of Figure 4.

We find that CryptDB supports most queries; the number of columns in the "needs plaintext" column, which counts columns that cannot be processed in encrypted form by CryptDB, is small relative to the number of columns encrypted. For OpenEMR, CryptDB does not support queries on certain sensitive fields that perform string manipulation (e.g., substring and lowercase conversions) or date manipulation (e.g., obtaining the day, month, or year of an encrypted date). However, if these functions were precomputed with the results added as standalone columns (e.g., by encrypting the three parts of a date separately), CryptDB would support these queries.

On the large sql.mit.edu trace, we found that CryptDB should be able to support operations over all but 1094 of the 128,840 columns observed in the trace. The "in-proxy processing" shows analysis results where we assumed that the proxy can perform some lightweight operations on the results returned from the DBMS server. Specifically, this includes operations that are not needed to compute the set of resulting rows, or to aggregate rows: that is, expressions that do not appear in a `WHERE`, `HAVING`, or `GROUP BY` clause, or in an `ORDER BY` clause with a `LIMIT`, and are not aggregate operators. With in-proxy processing, CryptDB should be able to process queries over encrypted data over all but 571 of the 128,840 columns, thus supporting 99.5% of the columns.

Of those 571 columns, 222 use a bitwise operator in a `WHERE` clause or perform bitwise aggregation, such as the Gallery2 application, which uses a bitmask of permission fields and consults them in `WHERE` clauses. Rewriting the application to store the permissions in a different way would allow CryptDB to support such operations. The other 205 columns perform string processing in the `WHERE` clause, such as comparing whether lowercase versions of two strings match. Storing a keyed hash of the lowercase version of each string for such columns, similar to the JOIN-ADJ scheme, could support case-insensitive equality checks for ciphertexts. Seventy-six columns are involved in mathematical transformations in the `WHERE` clause, such as manipulating dates, times, scores, and geometric coordinates. Forty-one columns invoke the `LIKE` operator with a column reference for the pattern; this is typically used to check a particular value against a table storing a list of banned IP addresses, usernames, URLs, etc. Such a query can also be rewritten if the data items are sensitive.

5.2. Security evaluation

To understand the amount of information that would be revealed to the adversary in practice, we examine the steady-state onion levels of different columns. To quantify the level of security, we define the MinEnc of a column to be the weakest onion encryption scheme exposed on any of the onions of a column when onions reach a steady state (i.e., after the application generates all query types, or after running the whole trace). We consider RND and HOM to be the strongest schemes, followed by SEARCH, followed by DET and JOIN, and finishing with OPE, which is the weakest scheme. For example, if a column has onion *Eq* at RND, onion *Ord* at OPE, and onion *Add* at HOM, the MinEnc of this column is OPE.

The right side of Figure 4 shows the MinEnc onion level for our applications and query traces. We see that *most fields remain at RND,* which is the most secure scheme, meaning that CryptDB leaks virtually nothing about most of the columns. We believe this is a strong indication that CryptDB achieves high security for practical applications. For example, OpenEMR has hundreds of sensitive fields describing the medical conditions and history of patients, but most of these fields are just inserted and fetched, and are not used in any computation, so they remain at RND. A number of fields also remain at DET, typically to perform key lookups and joins. Note that if the values encrypted with DET are distinct, DET is as secure as RND. OPE, which leaks order, is used the least frequently, and mostly for fields that are

Figure 4. Steady-state onion levels for database columns required by a range of applications and traces. "Consider for encryption" indicates the columns that should be encrypted: as explained in Section 5, these are the columns deemed sensitive for the four applications, and all columns for the two traces. "Needs plaintext" indicates the number of columns that should be encrypted, but for which CryptDB cannot execute the application's queries over encrypted data. MinEnc is defined in Section 5.2.

	Consider	Needs	Non-plaintext cols. with MinEnc:		
Application	for enc.	plaintext	RND/SEARCH	DET	OPE
phpBB	23	0	21	1	1
HotCRP	22	0	19	1	2
grad-apply	103	0	95	6	2
OpenEMR	566	7	528	12	19
TPC-C	92	0	65	19	8
Trace from sql.mit.edu	128,840	1094	80,403	34,212	13,131
...with in-proxy processing	128,840	571	84,406	35,350	8513

marginally sensitive (e.g., timestamps and counts of messages). This data demonstrates the importance of CryptDB's adjustable security: it provides a significant improvement in confidentiality over revealing all encryption schemes to the server.

For the sql.mit.edu trace, approximately 6.6% of the columns were at OPE even with in-proxy processing; the other encrypted columns remain at DET or above. Out of the columns that were at OPE, ~60% are used in an `ORDER BY` clause with a `LIMIT`, ~55% are used in an order comparison in a `WHERE` clause, and ~4% are used in a `MIN` or `MAX` aggregate operator (some of the columns are counted in more than one of these groups). It would be difficult to perform these computations in the proxy without substantially increasing the amount of data sent to it.

5.3. Performance evaluation

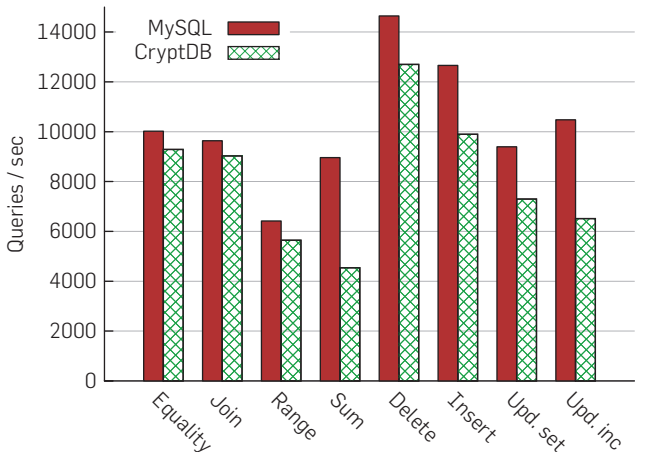
To evaluate the performance of CryptDB, we used a machine with two 2.4 GHz Intel Xeon E5620 4-core processors and 12 GB of RAM to run the MySQL 5.1.54 server, and a machine with eight 2.4 GHz AMD Opteron 8431 6-core processors and 64 GB of RAM to run the CryptDB proxy and the clients. The two machines were connected over a shared Gigabit Ethernet network. The higher-provisioned client machine ensures that the clients are not the bottleneck in any experiment. All workloads fit in the server's RAM.

5.3.1. TPC-C

We compare the performance of a TPC-C query mix when running on an unmodified MySQL server versus on a CryptDB proxy in front of the MySQL server. We warmed up CryptDB on the query set so that there are no onion adjustments during the TPC-C experiments. The server spends 100% of its CPU time processing queries.

We consider two important metrics: database server throughput (number of queries per second that the server can process) and latency (time interval from when the application issues a query to when it receives the result).

Figure 5. Throughput of different types of SQL queries from the TPC-C query mix running under MySQL and CryptDB. "Upd. inc" stands for `UPDATE` that increments a column, and "Upd. set" stands for `UPDATE` that sets columns to a constant.



The throughput with CryptDB was 26% lower than that with plain MySQL on TPC-C. We believe this overhead is modest considering the gains in confidentiality. To understand the sources of CryptDB's overhead, we measure the server throughput for different types of SQL queries seen in TPC-C, on the same server, but running with only one core enabled. Figure 5 shows the results for MySQL and CryptDB. The results show that CryptDB's throughput penalty is the greatest for queries that involve a `SUM` (half the throughput) and for incrementing `UPDATE` statements (1.6 × less throughput); these are the queries that involve HOM additions at the server. For the other types of queries, which form a larger part of the TPC-C mix, the throughput penalty is lower.

To understand the latency introduced by CryptDB, we measure the server and proxy processing times for the same types of SQL queries as above. The server latency is 0.12 ms, which is a 20% increase over the 0.10 ms latency of plain MySQL, which we consider to be small. The proxy adds an average of 0.60 ms to a query; of that time, 24% is spent on `mysql-proxy`, 23% is spent on encryption and decryption, and the remaining 53% is spent parsing and processing queries. The cryptographic overhead is relatively small because most of our encryption schemes are efficient. OPE and HOM are the slowest, but we performed two optimizations: pre-computing randomness to speed up encryption for HOM, and caching ciphertexts for OPE. Without these optimizations, the proxy latency would have been 10.7 ms on average in our experiments, which is significantly higher.

5.3.2. phpBB

We also evaluated the performance of CryptDB on phpBB, an open-source Web forum application. We measured the HTTP request processing throughput of a phpBB server using both CryptDB and a standard MySQL database. We encrypted only the sensitive fields as shown in Figure 4. We found that CryptDB reduced throughput by only 14.5%.

6. RELATED WORK

Search and queries over encrypted data. Cryptographic tools for performing keyword search over encrypted data have been proposed (e.g., Song et al.[22] which we use to implement SEARCH). When applied to processing SQL on encrypted data, these techniques suffer from some of the following limitations: certain basic queries are not supported or are too inefficient (especially joins and order checks), they require significant client-side query processing, users either have to build and maintain indexes on the data at the server or have to perform sequential scans for every selection/search, and implementing these techniques requires unattractive changes to the innards of the DBMS.

Some researchers have developed prototype systems for subsets of SQL, but they achieve lower security, require a significant DBMS rewrite, and rely on client-side processing. For example, Hacıgümüş et al.[10] heuristically split the domain of possible values for each column into partitions, storing the partition number unencrypted for each data item, and rely on extensive client-side filtering of query results.

Untrusted servers. SUNDR[14] uses cryptography to provide privacy and integrity in a file system on top of an untrusted file server. Using a SUNDR-like model, SPORC[5] shows how to build low-latency applications, running mostly on the clients, without having to trust a server. However, existing server-side applications that involve separate database and application servers cannot be used with SPORC unless they are rewritten as distributed client-side applications. Many applications are not amenable to such a structure.

Companies like Navajo Systems and Ciphercloud provide a trusted application-level proxy that intercepts network traffic between clients and cloud-hosted servers (e.g., IMAP), and encrypts sensitive data stored on the server. In comparison, CryptDB supports a richer set of operations (most of SQL) and provides better security.

Disk encryption. Various commercial database products, such as Oracle's Transparent Data Encryption,[16] encrypt data on disk, but decrypt it to perform query processing. As a result, the server must have access to decryption keys, and an adversary compromising the DBMS software can gain access to the entire data.

Software security. Many tools help programmers either find or mitigate mistakes in their code that may lead to vulnerabilities, including static analysis tools like UrFlow,[4] and runtime tools like Resin.[23] In contrast, CryptDB provides confidentiality guarantees for user data even if the adversary gains complete control over the application and database servers. These tools provide no guarantees in the face of this threat, but in contrast, CryptDB cannot provide confidentiality in the face of vulnerabilities that trick the user's client machine into issuing unwanted requests (such as cross-site scripting or cross-site request forgery vulnerabilities in Web applications). As a result, using CryptDB together with these tools should further improve application security.

Query integrity. CryptDB does not ensure that the query results from the server are correct, but most existing techniques for SQL query integrity can be integrated into CryptDB because CryptDB allows relational queries on encrypted data to be processed just like on plaintext.

7. CONCLUSION

We presented CryptDB, the first practical system that can execute a wide range of SQL queries on encrypted data. Using SQL-aware adjustable encryption with multiple onions, CryptDB provides a strong level of confidentiality in the face of two significant threats confronting database-backed applications: compromises to the DBMS server by a passive adversary, and arbitrary compromises to the application server and the DBMS. CryptDB requires no changes to the internals of the DBMS. Our evaluation shows that CryptDB successfully handles a wide range of queries observed in practice, with a modest performance overhead. CryptDB's Website (including papers and source code) is at http://css.csail.mit.edu/cryptdb/.

Acknowledgments

We thank everyone who helped with the original paper,[18] and Alon Halevy and the *Communications* staff for helping improve this paper. This work was supported by the NSF (IIS-1065219 and CNS-0716273) and by Google.

References

1. Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A. Order-preserving symmetric encryption. In *EUROCRYPT* (2009).
2. Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *Advances in Cryptology (CRYPTO)* (2011).
3. Chen, A. GCreep: Google engineer stalked teens, spied on chats. *Gawker* (2010). http://gawker.com/5637234/.
4. Chlipala, A. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (2010).
5. Feldman, A.J., Zeller, W.P., Freedman, M.J., Felten, E.W. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (2010).
6. *FIPS 197*. Advanced Encryption Standard (AES). U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, 2011.
7. Gentry, C. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing* (2009).
8. Gentry, C., Halevi, S., Smart, N.P. Homomorphic evaluation of the AES circuit. Cryptology ePrint Archive, Report 2012/099, 2012.
9. Goldreich, O. Foundations of Cryptography: Volume I Basic Tools, Cambridge University Press, 2001.
10. Hacıgümüş, H., Iyer, B., Li, C., Mehrotra, S. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of ACM SIGMOD* (2002).
11. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th Usenix Security Symposium* (2008).
12. Halevi, S., Rogaway, P. A tweakable enciphering mode. In *Advances in Cryptology (CRYPTO)* (2003).
13. Homeland Security News Wire. Data breaches compromise nearly 8 million medical records, 2011.
14. Li, J., Krohn, M., Mazières, D., Shasha, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (2004).
15. NIST. National Vulnerability Database. http://nvd.nist.gov.
16. Oracle Corporation. Oracle advanced security. http://www.oracle.com/technetwork/database/options/advanced-security/.
17. Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT* (1999).
18. Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011).
19. Popa, R.A., Zeldovich, N. Cryptographic treatment of CryptDB's adjustable join. Technical Report MIT-CSAIL-TR-2012-006, MIT Computer Science and Artificial Intelligence Laboratory, 2012.
20. Quinn, B., Arthur, C. Playstation network hackers access data of 77 million users. The Guardian, 2011.
21. Shi, E., Bethencourt, J., Chan, H., Song, D., Perrig, A. Multi-dimensional range query over encrypted data. In *Proceedings of the IEEE Symposium on Security and Privacy* (2007).
22. Song, D.X., Wagner, D., Perrig, A. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy* (2000).
23. Yip, A., Wang, X., Zeldovich, N., Kaashoek, M.F. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (2009).

Raluca Ada Popa, Catherine M.S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan, Computer Science and Artificial Intelligence Lab, M.I.T., Cambridge, MA.